API設計の基礎

初版第5刷

柴田 芳樹 著

2017年8月30日

まえがき

1978 年に大学で初めてコンピュータに触れてから様々なソフトウェア開発に従事してきました。しかし、大学での 6 年間や社会人となってからも、非効率なソフトウェア開発をしてきたのが現実です。C 言語で実装を隠蔽することを覚えたのは、社会人となってから 7、8 年過ぎてからでしたし、防御的プログラミングを始めたのは何と社会人となってから 15 年も過ぎていました。

本書では、C 言語、C++言語、Java 言語を中心として、日々のソフトウェア開発の中で意識して注意を払うべきことをまとめたものです。それらを実践するのとしないとでは、日々の開発の生産性が大きく左右されます。残念ながら、それらを教えられたり、指導されながらソフトウェア開発を行っているソフトウェアエンジニアは少ないです。

私自身も、特に誰かに教えてもらったのではなく、自分で考えたり、書籍を通して学んだりしました。しかし、最初にきちんと学んでいれば、もっと良いソフトウェア作りができたと思っています。

公開 API とは

外部へ公開する API（*Application Programming Interface*）は、以下の要件を備えている必要があります。

- 正常な動作が仕様に明確に定義されているだけでなく、関数、メソッド、コンストラクタなどのパラメータに不正な値が渡された場合の仕様が明確に定義されている。
- 利用者（クライアント）側の立場に立った分かりやすく、使いやすい API となっている。
- 誤った使い方ができないようになっている。
- API が提供する機能を実現するための実装詳細が隠蔽されており、実装詳細をクライアントから直接呼び出したり、参照したりすることができない。
- 実装詳細が変更されても、クライアントのソースコードをリコンパイルする必要がない。

逆に、悪い API の例としては、以下のようなことが挙げられます。

- パラメータに不正な値が渡された場合の動作が、記述されていない。
- 継承されることを意図していないのに、継承できるようになっている。
- 継承可能なクラスにおいて、継承のための文書化が行われていない。
- 関数名、クラス名、インタフェース名、メソッド名などの名前付けが悪い。その結果、名前から処理内容や責務が不明だったり、使用方法が分からない。

- 実装詳細である関数、クラス、メソッドなどが、クライアントから直接利用可能となっている。そのため、一度利用されてしまうと、その機能を永久にサポートしなければならない。
- 実装詳細が変更されるごとに、クライアントのソースコードをすべてリコンパイルしなければならない（C/C++）。
- クライアントのソースコードをコンパイルするために、クライアントには関係のないヘッダーファイルが多数 `include` され、不必要な依存関係が存在する（C/C++）。

本書の構成

本書は、以下の章から構成されています。

第 1 章「C/C++での実装の隠蔽」では、ライブラリーなどの API（*Application Programming Interface*）設計において、その実装を隠蔽する方法について説明します。C 言語による隠蔽、C++言語による隠蔽について同じ題材を使用して説明します。

第 2 章「Java での実装の隠蔽」では、パッケージを単位とした API の考え方に基づく実装の隠蔽について説明します。

第 3 章「防御的プログラミング」では、防御的にプログラミングすることの重要性と、その方法について説明します。

C 言語、C++言語、Java 言語の入門書ではありませんので、読者がこれらの言語に慣れ親しんでいる必要があります。どれかを知らない場合には、慣れ親しんでいる言語に関する解説だけを読まれても良いです。

一部のソースコードは示すことなく、読者への宿題として練習問題を用意してありますので、実際にプログラミングしてみてください。

本書は、私自身がソフトウェア開発を行いながら学んだことをまとめたものであり、その内容には偏りがあるかもしれません。しかし、本書を通して、みなさんが私と同じような遠回りをすることなくソフトウェア開発をされることの助けとなれば幸いです。

謝辞

本書の内容に関して、誤植の指摘、質問、助言をくださり、内容の改善に寄与してくださった、渡邉 剛、木村 順、松村 亮治、里田 和敏、浜野 義丈、前村 浩一郎の各氏に感謝します。

また、佐竹 雅紀、志甫 裕一、田村 純一、藤井 俊英、藤井 英夫、吉川 悟史、露崎 規彦の各氏には、本書を題材とした勉強会で気付かれた誤植を指摘いただいたことに感謝します。

柴田 芳樹

2016 年 12 月

目次

第 1 章

C/C++での実装の隠蔽

何らかの機能を提供するライブラリーを開発する場合、そのライブラリーの公開 API（*Application Programming Interface*）を、ライブラリーのクライアントへ提供することになります。C 言語や C++ 言語であれば、ヘッダーファイルとして提供することになります。Java 言語であれば、クラスやインタフェースをコンパイルしたものを提供することになります。

公開 API に対してコンパイルされたクライアントのコードをリコンパイルすることなく、ライブラリーの実装者は自由に実装方法を変更できる必要があります。バグ修正、実装方法の変更、あるいは、パフォーマンスチューニングのための修正を行うために、クライアントにリコンパイルを依頼できるとは限りません。あるいは、1 つの開発組織内であってもすべてをリコンパイルすることなく一部のライブラリーの実装を変更してテストしたい場合もあります。

C 言語や C++言語で実装を隠蔽するとは、公開 API となるヘッダーファイルの中に実装の詳細を含めない設計を行うことです。実装の詳細を含めるとは、実装に使用されるデータ構造が含められていたり、そのデータ構造をコンパイルするために他の実装用のヘッダーがインクルードされていたりすることを指します。クライアントにとって全く関係のないヘッダーがインクルードされていると、それだけでコンパイル時間を増大させます。また、実装用のヘッダーの内容を見ることで、クライアントが実装用の関数を直接呼び出したり、実装用のデータ構造を直接操作することを可能にします。

本章では、簡単なスタックを例として、実装の隠蔽について説明していきます。

1.1 抽象データ型

実装の隠蔽について説明する前に、重要な概念である**抽象データ型**（*Abstract Data Type*）について説明します。抽象データ型とは次の要件を満たしているものです。

- データとそれに対する操作の集まりをまとめている
- データ構造と操作の実装を隠蔽している

C 言語は、言語仕様として抽象データ型を直接サポートしていません。C++言語が登場する前に、抽象データ型を言語仕様としてサポートしている言語としては、Mesa、Ada、Modula-2 などがありましたが、どの言語も日本では広く普及はしていません。

1.1.1 人間の思考

ソフトウェア開発において、どのような分析ができるかは、その人がどのような設計ができるかに影響を受けます。さらに、どのような設計ができるかは、どのようなプログラミング言語を使用できるかに影響を受けます。たとえば、次の通りです。

- アセンブリ言語しか使ったことがない人は、構造化プログラミングをしない
- C 言語しか使ったことがない人は、抽象データ型を設計しない
- C 言語しか使ったことがない人は、オブジェクト指向プログラミングをしない

このように使用している言語がサポートする概念だけを使用してプログラミングすることを「*Programming in a language*」（言語の**中で**のプログラミング）[*1] と言います。プログラミング言語が、プログラマーの思考にどのような影響を与えるかについて、Eric Merritt 氏は次のように述べています。

> 様々な方法で、プログラミング言語は、パラカス（Paracas）での幼児の頭蓋骨、漢（Han）女性の足、カレン・パダウン（Karen Paduang）女性の首を形成するために使われる道具にかなり似たように振る舞います。頭蓋骨を形成する代わりに、問題についての考え方、アイデアの生まれ方、それに、それらのアイデアが特定の問題に適用される方法を、プログラミング言語は形作ります。たとえば、初期のバージョンの Fortran （Fortran 77 以前） でしかコードを書いたことがなければ、おそらく再帰呼び出しの存在を知らないでしょう。また、Haskell でしかコードを書いたことがなければ、命令的なスタイルのループについてはほとんど知ることはないでしょう。
>
> Eric Merritt、“*The Shape of Your Mind*” [*2]

一方で、言語にとらわれず、行いたい設計を与えられた言語で実現することで、次のようなことを行うことを「*Programming into a language*」（言語の**中へ**のプログラミング）[*3] と言います。

- アセンブリ言語で構造化プログラミングを行う
- C 言語で抽象データ型の設計を行う
- C 言語でオブジェクト指向プログラミングを行う

Steve McConnell 氏は、次のようにまとめています。

> 重要なプログラミングの原理のほとんどは、言語の種類ではなく、言語を使用する方法に依存する。使用したい構造が言語に含まれていない、あるいは何らかの問題が起きやすいという場合には、それらを補正できるかを試してみよう。コーディング規約、標準、クラスライブラリなどの補強を独自に考案してみよう。
>
> Steve McConnell、『コードコンプリート第 2 版』

[*1] Steve McConnell 著、『コードコンプリート第 2 版 上』（日経 BP ソフトプレス）の 4.3 節。

[*2] `http://erlangish.blogspot.com/2007/05/shape-of-your-mind.html`
訳は、『アプレンティスシップ・パターン』（オライリー・ジャパン）からの引用。

[*3] Steve McConnell 著、『コードコンプリート第 2 版 下』（日経 BP ソフトプレス）の 34.4 節。

では、C 言語での実装の隠蔽を通して、C 言語での抽象データ型の設計を見ていきます。

1.2 C 言語によるスタックの実装例

スタックの基本動作は、プッシュ（*push*）とポップ（*pop*）です。たとえば、int 値を入れるスタックを考えてみると次のような操作が考えられます。

リスト 1-1 stack.h **スタックが 1 つしかない**

```
void stack_initialize(int initialCapacity);
void stack_destroy(void);
void stack_push(int data);
int  stack_pop(void);
```

stack_initialize 関数は、スタック操作をするための初期化を行います。stack_destroy 関数は、スタックの利用を終了します。stack_push 関数はプッシュ操作を行い、stack_pop 関数はポップ操作を行います。この場合、スタックの実装は、クライアントから完全に隠蔽されています。

しかし、このスタックの実装では、システムに 1 つのスタックしか存在しないことになり、かなり不便です。したがって、多くのスタックを利用可能にするために、次のようにスタックの API をヘッダーファイル（stack.h）に宣言します。

リスト 1-2 stack.h **複数のスタックが可能**

```
struct stack {
    int *elements;
    int capacity;
    int size;
};

void stack_initialize(struct stack *stack, int initialCapacity);
void stack_destroy(struct stack *stack);
void stack_push(struct stack *stack, int data);
int  stack_pop(struct stack *stack);
```

そして、実装ファイル（stack.c）[*4] は、次のようになります。

[*4] 本書のコード例では、malloc でのメモリの割り当ての失敗を検査していません。メモリ不足への対処は、個々の箇所で行う前に、システムとしてどのように統一的に取り扱うかを決める必要があります。したがって、本書ではメモリ不足の検査は行わないコードとなっています。

リスト 1-3 stack.c **スタックの実装**

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "stack.h"

void stack_initialize(struct stack *stack, int initialCapacity) {
    if (stack == NULL) {
        assert(0 && "Illegal stack pointer");
        return;
    }

    if (initialCapacity <= 0) {
        assert(0 && "Illegal initial capacity");
        return;
    }

    stack->elements = (int *) malloc(initialCapacity * sizeof(int));
    stack->capacity = initialCapacity;
    stack->size = 0;
}

void stack_destroy(struct stack *stack) {
    if (stack == NULL) {
        assert(0 && "Illegal stack pointer");
        return;
    }

    free(stack->elements);
}

void stack_push(struct stack *stack, int data) {
    if (stack == NULL) {
        assert(0 && "Illegal stack pointer");
        return;
    }

    if (stack->capacity == stack->size) {
        int newCapacity = stack->capacity * 2 + 1;
        int *newElements = (int *) malloc(newCapacity * sizeof(int));
        memmove(newElements, stack->elements, sizeof(int) * stack->size);
        free(stack->elements);
        stack->elements = newElements;
        stack->capacity = newCapacity;
```

```
        }

        stack->elements[stack->size++] = data;
    }

    int stack_pop(struct stack *stack) {
        if (stack == NULL) {
            assert(0 && "Illegal stack pointer");
            return 0;
        }

        if (stack->size == 0) {
            assert(0 && "Empty stack");
            return 0;
        }

        return stack->elements[--stack->size];
    }
```

これで、クライアントでは、次のようにスタックの実体を用意して、初期化してからスタックを利用することができます。

リスト 1-4　スタックを利用するクライアント側コード例（その 1）

```
#include "stack.h"
...
struct stack stack;

stack_initialize(&stack, 1);
stack_push(&stack, 1);
...
int data = stack_pop(&stack);
...
stack_destroy(&stack);
```

あるいは、malloc/free を使用すると次のようになります。

リスト 1-5 スタックを利用するクライアント側コード例（その 2）

```
#include "stack.h"
...
struct stack *stack = (struct stack *) malloc(sizeof(struct stack));

stack_initialize(stack, 1);
stack_push(stack, 1);
...
int data = stack_pop(stack);
...
stack_destroy(stack);
free(stack);
```

このスタックの機能は、不十分です。たとえば、スタックが空の場合にポップ操作を行うと assert 文によるアサーションになるか、assert 文が無効になっていれば 0 が返ってきます。したがって、ポップ操作を行う前に、スタックが空か否かを判定する必要があるのですが、そのための関数が提供されていません。

クライアントがこのような状況に直面した場合には、スタックの作者に関数の追加を要請して提供してもらうのが正しい対処方法です。なぜなら、スタックの API は、機能が不十分だからです。ところが、幸いに（？）、スタックのデータ構造が stack.h に記述されており、リスト 1-4 で示されているクライアントは、次のようにもコードを書くことができてしまいます。

リスト 1-6 クライアントによる誤った実装参照

```
if (stack.size != 0) {
    int data = stack_pop(&stack);
    ....
}
```

一度、このようなクライアントのコードが書かれてしまうと、スタックのデータ構造を変更することが困難となります。変更するとクライアントのコードがコンパイルできなくなってしまう可能性があるからです。スタックの作者は「直接データ構造を参照されたり操作されたりすることは想定していなかった」と言い訳できますが、それも後の祭りです。**問題は、クライアントが参照したり操作して欲しくない実装の詳細を公開してしまったことです。**

また、構造体 struct stack の大きさは、クライアントのコードをコンパイルした時に埋め込まれます。リスト 1-4 の場合には、struct stack の領域を確保する大きさとして埋め込まれます。リスト 1-5 の場合には、malloc 関数の引数として埋め込まれます。

小規模な開発においても、実装の詳細が公開されていることによる問題は発生します。ライブラリーの担当者がデータ構造を変更したにも関わらず、クライアントのコードをリコンパイルせずにリンクす

ると、正しく動作しなかったりします。あるいは、クライアントのコードをリコンパイルしようとしたらコンパイルエラーになることもあります。どちらの場合も、ライブラリーの担当者は実装の詳細を変更しただけと思っているだけかもしれませんが、現実にはこのような問題は発生します。

練習問題 1.1：公開 API に実装の詳細を入れて、クライアントをリコンパイルしなければならないのにリコンパイルすることなくリンクしたために発生した問題を調査したことがありますか。どのような現象の問題であったのか、調査にどのくらい時間を要したのかを説明しなさい。また、同じ問題を回避するためにどのような対処を行いましたか。

> 次の節で説明する方法は、マッキントッシュ用アプリケーションのプログラミングを覚えようとして、プログラミングの本を読んでいる時に学んだと記憶しています。1992 年前後だと思います。当時は、Xerox 社の PARC（Palo Alto Research Center）内のあるプロジェクトに従事しており、C 言語でプログラミングしていました。

1.3 C 言語による実装の隠蔽

スタックの例で実装の詳細を隠蔽するには、次のようにヘッダーファイル `stack.h` を定義します。

リスト 1-7 `stack.h` **実装を隠蔽したヘッダー**

```
struct stack; // forward declaration

struct stack *stack_create(int initialCapacity);
void stack_destroy(struct stack *stack);

void stack_push(struct stack *stack, int data);
int  stack_pop(struct stack *stack);
int  stack_peek(struct stack *stack);
int  stack_isEmpty(struct stack *stack);
```

一行目で構造体 `struct stack` への**前方宣言**（*forward declaration*）していることに注意してください。`stack_initialize` 関数の代りに、`stack_create` 関数を定義しています。また、スタックの先頭から要素を取り除くことなく返す `stack_peek` 関数とスタックが空か否かを返す `stack_isEmpty` 関数を追加しています。

このスタックを利用するクライアント側のコードは、次のようになります。

リスト 1-8 クライアント側のコード例

```
#include "stack.h"
...
struct stack *stack = stack_create(1);

stack_push(stack, 1);
stack_push(stack, 2);
...
if (!stack_isEmpty(stack)) {
    int data = stack_pop(stack);
    ...
}
...
stack_destroy(stack);
```

スタックの領域をクライアントが用意するのではなく、stack_create 関数により動的に生成しています。stack.h 内での前方宣言とこの動的生成により、クライアントのオブジェクトの中にはスタックを実装している構造体の大きさに関する情報は何も埋め込まれていません。

次に、スタックの実装を見ていきます。スタックの構造体（struct stack）の実体は、次のように実装用ヘッダーファイル（stackImpl.h）に定義します。

リスト 1-9 stackImpl.h 実装用ヘッダー

```
struct stack {
    int *elements;
    int capacity;
    int size;
};
```

スタックの実装ファイル（stack.c）は、次のように変更になります。

リスト 1-10 stack.c 実装ファイル

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "stack.h"
#include "stackImpl.h"
```

```
struct stack *stack_create(int initialCapacity) {
    if (initialCapacity <= 0) {
        assert(0 && "Illegal initial capacity");
        return NULL;
    }

    struct stack *stack = (struct stack *) malloc(sizeof(struct stack));

    stack->elements = (int *) malloc(initialCapacity * sizeof(int));
    stack->capacity = initialCapacity;
    stack->size = 0;
    return stack;
}

void stack_destroy(struct stack *stack) {
    if (stack == NULL) {
        assert(0 && "Illegal stack pointer");
        return;
    }

    free(stack->elements);
    free(stack);
}

/* 残りは省略 */
```

stack_create 関数内でスタックの構造体を動的に生成し、stack_destroy 関数内では最後にその構造体を解放しています。残りの関数は省略してあります。

このような方法で、公開 API となるヘッダーファイルに実装の詳細を入れることなく、そのヘッダーファイルとコンパイルされたライブラリーのオブジェクトだけをリリースすることで、クライアントはライブラリーの実装の詳細から完全に隔離されます。ライブラリーの実装者も、クライアントのコードをリコンパイルすることなく、自由に実装の詳細を変更することが可能になります。

ここで示した方法を使用することで、**抽象データ型**（*Abstract Data Type*）を C 言語でも容易に実現できます。

練習問題 1.2：リスト 1-6 で定義されている stack.h の機能を確認するテストプログラムを作成しなさい。そのテストプログラムを動作させるために、リスト 1-9 の stack.c を完成させなさい。

練習問題 1.3：スタックの実装を配列ではなく、リンクリスト（*linked list*）で書き直しなさい。その際に、stack_create 関数の引数を無視して構いません。リンクリスト版の実装を、練習問題 1.1 で作成したテストプログラムを利用して動作を確認しなさい。

1.3.1 関数以外の定義について

公開 API 用のヘッダーには、関数のプロトタイプ宣言以外にも、構造体の定義や、定数定義を入れることがあります。その場合も、あくまでも公開 API の一部としての構造体定義や定数定義となります。たとえば、API として定義している関数の引数として使用する構造体や定数などです。

言い換えると、実装でしか使用しない構造体や定数定義を公開 API 用ヘッダーファイルに含めないようにします。構造体などは、一度公開 API 用ヘッダーに定義してリリースすると、定義の変更が困難になりますので、慎重に検討して判断する必要があります。

1.4 C++言語によるスタックの実装例

C++言語では、クラスを宣言します。たとえば、Stack クラスは次のようになります。

リスト 1-11 Stack.h **公開 API クラス**

```
class Stack {
public:
    Stack(int initialCapacity);
    ~Stack();

    void push(int data);
    int pop(void);
    int peek(void);
    bool isEmpty(void);

private:
    int *elements;
    int capacity;
    int size;
};
```

C++言語は、public ラベルと private ラベルによって、クライアントがアクセスできるメンバー関数やフィールドを制御します。リスト 1-11 では、クライアントは Stack クラスのフィールドである elements、capacity、size にはアクセスできません。そして、クライアントは次のようにこのクラスを利用することができます。

リスト 1-12 クライアントのコード例

```
Stack *stack = new Stack(1);

stack->push(1);
stack->push(2);
...
int data = stack->pop();
...
delete stack;
```

実装の詳細である private フィールドは、クライアントからアクセスできませんので、問題が無いように見えます。でも、本当でしょうか。クライアントをリコンパイルすることなく、実装の詳細を変更できるでしょうか。

残念ながら C++言語では、クライアントのコードがコンパイルされる時に Stack クラスのインスタンスの大きさが計算され、インスタンス生成時に必要なメモリ量がクライアントのコードがコンパイルされた時に（new オペレータの第 1 引数として）埋め込まれてしまいます。そのために、private なフィールドを追加してインスタンスの大きさを変更すると、クライアントのコードをリコンパイルしないと古い大きさのままとなってしまいます。つまり、フィールドを追加すると、クライアントのコードをリコンパイルする必要があるということです。

> 1993 年 5 月に 4 年半の米国駐在を終えて日本に帰国した時に従事したのが 1996 年に製品化された *Fuji Xerox DocuStation IM 200* でした。開発言語は C 言語ではなく、C++言語でした。それまでは C++言語での開発経験がなく、その製品開発のために集められたエンジニアのほぼ全員が C++言語での開発経験がありませんでした。私自身が C++言語を学習してみて、悩んだのがこの節で述べた問題でした。この問題を解決してから開発に着手しないと大変なことになるという直感から、色々と調べながら悩んだ末に考え出したのは次節で述べる方法でした。

1.5 C++言語による実装の隠蔽

公開 API として含まれるヘッダーに、実装の詳細を一切入れなければ良い訳ですから、次のように宣言します。

リスト 1-13 Stack.h 実装の詳細を入れない（不完全）

```
class Stack {
public:
    virtual ~Stack();

    virtual void push(int data) = 0;
```

```
    virtual int  pop(void)      = 0;
    virtual int  peek(void)     = 0;
    virtual bool isEmpty(void)  = 0;
};
```

デストラクタを virtual 宣言し、それ以外のメンバー関数はすべて**純粋仮想関数**（*pure virtual function*）として宣言します。このように宣言することで、実装の詳細が何も記述されていないことになります。

しかし、この宣言には問題があります。それは、クライアントがインスタンスを生成できないことです。Stack クラスは**抽象クラス**（*Abstract Class*）になっていますので、new Stack() によるインスタンスを生成できません。クライアントがインスタンスを入手できるように、コンストラクタの代りに**static ファクトリーメソッド**（*static factory method*）を次のように追加します。

リスト 1-14　Stack.h　実装の詳細を入れない（完全版）

```
class Stack {
public:
    static Stack *createInstance(int initialCapacity);

    virtual ~Stack();

    virtual void push(int data) = 0;
    virtual int  pop(void)      = 0;
    virtual int  peek(void)     = 0;
    virtual bool isEmpty(void)  = 0;
};
```

リスト 1-14 の Stack クラスの定義（Stack.h）を公開 API とすることで、クライアントは次のようにコードを書くことができます。

リスト 1-15　クライアントのコード例

```
#include "Stack.h"

...
Stack *stack = Stack::createInstance(1);
...
stack->push(1);
stack->push(2);
...
```

```
int data = stack->pop();
...
delete stack;
```

コンストラクタが提供されていませんので、static ファクトリーメソッドである createInstance 関数を呼び出してインスタンスを取得していることに注意してください。

> 正直に言えば、この static ファクトリーメソッドは、1993 年には思い付きませんでした。では何を思い付いたかというと、コンストラクタの代りに、インスタンス生成のためのグローバルな関数を定義するというものです。そして、その関数名は、「new_クラス名」と命名するというものでした。static ファクトリーメソッドをいつ思い付いたのかは、記憶がはっきりしませんが、2000 年にはこの方法でライブラリーを設計していました。おそらく、1996 年夏から Java を学び始めたことで気付いたのだと思います。スコット・メイヤーズの『Effective C++ 第 3 版』（丸善出版）の項目 31 では「インタフェースクラス」として紹介されています。

では、次に実装を見ていきます。実装の詳細が入ったクラス（StackImpl.h）を次のように定義します。

リスト 1-16　StackImpl.h　実装用ヘッダーファイル

```
#include "Stack.h"

class StackImpl: public Stack {
public:
    StackImpl(int initialCapacity);
    ~StackImpl();

    void push(int data);
    int  pop(void);
    int  peek(void);
    bool isEmpty(void);

private:
    int *elements;
    int capacity;
    int size;
};
```

StackImpl クラスは、Stack クラスを継承しています。そして、private セクションに実装に必要なフィールドが宣言されています。これで Stack クラスの実装を次のように記述することができます。

リスト 1-17 Stack.cpp

```
#include "StackImpl.h"

Stack *Stack::createInstance(int initialCapacity) {
    return new StackImpl(initialCapacity);
}

Stack::~Stack() { }
```

実際の実装は StackImpl クラスで行われ、クライアントからは完全に隠蔽されていますので、自由に StackImpl クラスを変更できます。

1.3 節「C 言語による実装の隠蔽」では、生成と解放の両方の関数を提供していました。この節の C++言語の場合には、生成だけが隠蔽されており、解放に関しては delete オペレータをそのまま利用する例となっています。もし、メモリを獲得してインスタンスを生成して返す new オペレータが利用するメモリ領域と、クライアントが呼び出す delete オペレータがメモリを解放するメモリ領域が異なる場合には、解放のための static ファクトリーメソッドを提供する必要があります。

練習問題 1.4：Stack クラスのテストプログラムを作成しなさい。そのテストプログラムを利用して、**テストファースト**（*Test First*）方式で StackImpl クラスを実装（StackImpl.cpp）しなさい。

練習問題 1.5：リンクリストでスタックを実装して、練習問題 1.4 で作成したテストプログラムで動作を確認しなさい。

1.6 公開ヘッダーファイルの配置

C 言語および C++言語での実装の隠蔽で、注意しなければならないのは、ライブラリーの公開 API としてのヘッダーとしては、stack.h や Stack.h のみを公開し、決して stackImpl.h や StackImpl.h を公開しないことです。そのためには、実際のディレクトリー構成を上手く工夫する必要があります。たとえば、公開用ヘッダーはすべて include ディレクトリーの下に置いて、実装用ヘッダーと実装ファイルは src ディレクトリーの下に置くなどの工夫が必要です。その結果、リリースされるのはライブラリーのオブジェクトファイルと include ディレクトリーの下のヘッダーファイルだけになるように工夫が必要です。

レイヤー構成からなる規模が大きいシステムでは、公開ヘッダー用の include ディレクトリーだけでは不十分な場合があります。同一レイヤー内のモジュールに対して公開したいが、上位のレイヤーに対しては公開したくないということが起きます。この場合、同一レイヤー内の他のモジュールへ公開するヘッダーファイルを配置するために、たとえば、friend_include などのように公開ヘッダー用とは別のディレクトリーを用意すると良いです。

1.7 API の更新と不整合

ライブラリーに機能が追加されていく過程で従来の API が修正されて、クライアントのコードをリコンパイルが必要になる場合があります。たとえば、C++言語ならば純粋仮想関数である新たなメンバー関数を追加した場合には、クライアントのコードをリコンパイルする必要があります。そのような場合、リコンパイルされずに新しいライブラリーをリンクしてしまったりすると、それが原因で不具合が発生します。そのような不具合の調査には時間を要する場合が多いです。

今日、継続的インテグレーションを行っている同一プロジェクト内であれば、すべて自動的にリコンパイルされるために問題が発生することはありません。しかし、そのように自動的にすべてのクライアントのコードをリコンパイルできる環境が常に存在するとは限りません。そのためには、誤ったライブラリーのバージョンが使用されていることを実行時に早期に検出できる必要があります。

C++言語でバージョンの不整合を実行時に検出する方法の 1 つとして、リスト 1-18 に示すように公開 API となるヘッダーファイルにバージョン情報を宣言します。

リスト 1-18 Stack.h

```
class Stack {
public:
    enum { API_VERSION = 1 };

    static Stack *createInstance(int initialCapacity,
                                 int apiVersion = API_VERSION);
    virtual ~Stack();

    virtual void push(int data) = 0;
    virtual int  pop(void)      = 0;
    virtual int  peek(void)     = 0;
    virtual bool isEmpty(void)  = 0;
};
```

クライアントのコードをリコンパイルしなければならない修正を行った場合には、この API_VERSION の値を必ず増やします。この新たなヘッダーファイルでは、createInstance 関数に新たな引数を追加していますが、デフォルト値として API_VERSION を指定していますので、クライアントのコードでは引数 apiVersion を渡す必要はありません。つまり、従来通り、次のように初期化することになります。第 2 引数は、自動的にコンパイラが API_VERSION の値を渡すコードを生成してくれます。

```
Stack *stack = Stack::createInstance(1);
```

ライブラリーの実装側では、次のように apiVersion を検査します。

リスト 1-19 Stack.cpp

```
#include <assert.h>
#include "StackImpl.h"

Stack *Stack::createInstance(int initialCapacity, int apiVersion) {
    if (apiVersion != API_VERSION) {
        assert(false && "API Version Mismatched");
        return 0;
    }
    return new StackImpl(initialCapacity);
}

Stack::~Stack() { }
```

API_VERSION が 2 であるライブラリーをリリースしたとします。そうすると、リスト 1-19 の API_VERSION は 2 となっています。もし、API_VERSION が 1 である古いバージョンに対してコンパイルされたクライアントのコードが、リコンパイルされることなく新たなライブラリーをリンクして使用すると、実行時に apiVersion として 1 が渡されますので、バージョンの不整合を検出することになります。

練習問題 1.6：C 言語のライブラリーで同様なバージョン不整合を検出する方法を考えなさい。

練習問題 1.7：誤って古いライブラリーをリンクして、それが原因となる不具合をデバッグしたことがありますか。原因に気づくのにどのくらい時間を要しましたか。同じようなトラブルを回避する方法として、どのような再発防止方法を検討して導入しましたか。

1.8 まとめ

C 言語および C++言語での実装の隠蔽方法について説明しました。これらの言語では構造体やクラスの大きさが、コンパイル時に計算されて埋め込まれてしまいます。したがって、公開 API を設計する場合には、そのことを考慮する必要があります。Java 言語ではコンパイル時にインスタンスの大きさが計算されることはありませんので、本章で述べた内容は当てはまりません。

最近では、最初に学習する言語が Java 言語の人も多くなっています。C 言語や C++言語でプログラミングをしなければならなくなった場合には、本章の内容を参考にしてください。

1.9 補足：警告について

どの言語で開発する場合でも、コンパイラが発する警告を無視してはいけません。自分が書いたコードに対して、警告が出た場合には、その警告を無視しても良いか否かを判断できるかもしれません。し

かし、他人から引き継いだコードをコンパイルしただけで、多くの警告が出ているとしたら、自分が新たに追加・修正したコードによる警告には気づかないかもしれません。

GNU のコンパイラーを使用するのであれば、-Wall オプションを指定することで、コンパイラーが警告と見なすすべての種類の問題点を表示してくれます。しかし、単に警告が表示されるだけでは、見落とす可能性も高かったり、あるいは、無視するエンジニアもいたりするかもしれません。

警告を取り除くには、その警告の意味をきちんと理解する必要があります。最近のコンパイラーは、printf の書式とその後に続く引数の型が異なる場合も警告を出します。その場合に、単に警告を無くすように型キャストすると正しく動作しなかったり、期待する表示がされなかったり、あるいはセグメンテーションフォルトが発生するかもしれません。

警告を取り除くことをプロジェクトで強制するには、コンパイルオプションとして、-Werror も同時に指定するようにします。-Werror を指定することで、警告が 1 つでもあるとコンパイルエラーと同じ扱いとなり、コンパイルが失敗します。

-Wall -Werror の指定は、プロジェクトが始まった時点から標準で指定するようにしなければなりません。プロジェクトの終盤に指定を強制しようとすると、その時点で多くの警告が存在して、導入そのものを躊躇することになります。実際、大規模なソフトウェア開発で全く導入されていないために修正しきれないほどの数の警告が存在するプロジェクトや、新規プロジェクトでありながら導入しないで開発を始めたために終盤での導入を躊躇するプロジェクトを見てきました。そのようなプロジェクトでは、障害を多く出すだけでなく、エンジニア自身が警告に対して無頓着となる弊害も生み出すことになります。

第 2 章

Java での実装の隠蔽

Java 言語での開発は、C 言語や C++言語での開発とは異なります。その違いの 1 つは、クラスのインスタンスの大きさが、クライアントのコードをコンパイルした時点で、クライアント側のコンパイル済みオブジェクトへ埋め込まれるか否かという点です。第 1 章「C/C++での実装の隠蔽」で述べた方法は、C/C++言語の場合には、インスタンスの大きさがクライアントのコードをコンパイルした結果のオブジェクトファイル内（.o ファイル）に埋め込まれれてしまうことを回避する方法とも言えます。一方、Java 言語では、そのような点を考慮する必要はありません。

Java 言語で考慮すべきことは、どの単位で公開 API を設計するかということです。Java 言語の場合は、基本的に**パッケージ単位で考える**必要があります。一方で、パッケージ単位の公開 API の設計にも限界があり、運用でカバーしなければならない点もあります。本章では、それらに関して説明します。

2.1 アクセス修飾子の基本

Java 言語には、アクセス修飾子として以下の 4 種類があります。

- `public` 他のパッケージからアクセス可能
- **パッケージ・プライベート** [*1] 同じパッケージからのみアクセス可能
- `protected` 他のパッケージのサブクラスからアクセス可能。また、同じパッケージからアクセス可能。
- `private` クラス内のメンバー [*2] からのみアクセス可能

トップレベルのクラス（enum を含む）やインタフェースに対するアクセス修飾子は、public かパッケージ・プライベートの 2 種類しかありません。クラスのメンバーに対するアクセス修飾子としては、上記の 4 種類が適用されます。

[*1] アクセス修飾子を何も書かない場合

[*2] クラスのメンバーとは、以下のものを指します。
・ フィールド
・ メソッド
・ ネストしたクラスとネストしたインタフェース
コンストラクタは含まれないことに注意してください。詳細については、『プログラミング言語 Java 第 4 版』2.1.1 節「クラスのメンバー」(p.36) を参照してください。

2.2 パッケージ単位の公開 API

Java 言語での開発では、機能をパッケージ単位に分割します。個々のパッケージは次の 2 種類から構成されます。

- 公開 API としてのクラスやインタフェース
- 実装詳細としてのクラスやインタフェース

パッケージを作成する場合には、パッケージの**公開 API** とその**実装詳細**を明確に区別する必要があります。残念ながら、多くの Java 関連の入門書では、このパッケージ単位の API 設計ということは十分に説明されておらず、公開 API と実装詳細を区別せずに設計されているシステムが多くあります。

C 言語や C++言語では、パッケージ・プライベートと同等のアクセス制限は、言語仕様として存在しません。それらの言語でのプログラミング経験しかない場合に、「**パッケージ単位の API 設計**」という概念は容易に理解できなかったりします。その結果、何も考えずに、すべてが `public` 宣言されたトップレベルのクラスやインタフェースが作成されることになります。

2.2.1 公開 API としてのクラスやインタフェース

パッケージの公開 API を構成する要素は、以下の通りです。

- `public` 宣言されたトップレベルのインタフェース。
- `public` 宣言されたトップレベルのクラス内に、`public` あるいは `protected` 宣言されたメンバーとコンストラクタ。
- `public` 宣言されたトップレベルの `enum` 内に、`public` 宣言されたメソッドやフィールド。
- `Serializable` インタフェースを実装しているクラスは、シリアライズされるフィールド。

`public` 宣言されていないトップレベルのクラスやインタフェースは、パッケージ外からはアクセスできませんので、公開 API の実装詳細となります。`public` 宣言されたトップレベルのクラスであっても、そのメンバーのすべてが公開 API を構成するとは限りません。公開 API ではないく実装詳細のメンバーやコンストラクタは、すべてパッケージ・プライベートか `private` 宣言しなければなりません。ネストしたクラスやインタフェースに対しても同様です。

シリアライズ可能なクラスのフィールドが公開 API と見なされることに関しては本書の範疇外です。詳しくは、『Effective Java 第 2 版』の第 11 章「シリアライズ」を参照してください。

2.2.2 実装詳細としてのクラスやインタフェース

公開 API の機能を実装するためだけのクラスやインタフェースを、トップレベルで `public` 宣言してはいけません。同様に、公開 API としてのクラス内に機能を実装するために定義されたネストしたクラスやインタフェースも `public` 宣言してはいけません。

それらは、公開 API の機能を実装するためのものであり、パッケージ外から使用されることを想定していない訳ですから、**最初からパッケージ外からアクセスできないようにしておくべき**です。本来、パッケージ外からアクセスされることを想定していないとしても、アクセスできる場合にはアクセスされてしまいます。そのような場合に、「パッケージ外からアクセスしてはならない」と、ドキュメンテーションに記述しても無駄です。そのようなドキュメンテーションを書くよりは、最初からアクセスできないようにするのが基本です。

ところが、Eclipse などの IDE を使用して開発している人の中には、ツールが生成したコードに対して無頓着だったりします。その結果、すべてのクラスが public 宣言されていたりします。あるいは、公開 API とその実装詳細を区別することを意識しながらクラスを作成していないために、無条件に public 宣言していたりします。

練習問題 2.1：今までの Java 言語によるプログラミングで、パッケージ単位で API を設計してきましたか。過去に書いたコードを見直して、不注意に他のパッケージに公開されているクラスやメソッドなどがないかを調査しなさい。

2.2.3 実装詳細の誤った公開 API 化の例

公開 API となるクラスは、そのクラスが提供する機能を実現するための実装詳細を公開してはいけないのですが、次のように安易に公開してしまっているコードを見かけることが多いです。

リスト 2-1 実装詳細を公開している

```
public final class X extends Thread {
    ...
    public X() {
        ...       // 他の初期化処理
        start();  // スレッドの開始
    }

    public void run() {
        ...  // スレッドでの処理
    }
}
```

この X クラスは、Thread クラスを継承しているために、run メソッドを public 宣言して、公開してしまっています。その結果、クライアントが run メソッドを直接呼び出すことが可能になってしまっています。

このような例は、Java の標準ライブラリにも存在しています。たとえば、java.util.Stack クラスは、java.util.Vector クラスを継承しているために、Vector クラスが提供している操作を行うことが可能です。また、java.util.Properties クラスは、java.util.Hashtable クラスを継承してい

るため、Hashtable クラスが提供している機能が公開されています。

このような場合、他のクラスを継承するのではなく、内部で使用する設計にする必要があります。リスト 2-1 を書き直すとリスト 2-2 のようになります。

リスト 2-2 実装詳細を隠蔽

```
public final class X {
    ...
    public X() {
        ...       // 他の初期化処理
        Thread t = new Thread(() -> {
            ...  // スレッドでの処理
        });
        t.start();
    }
}
```

リスト 2-2 は、Java 8 で導入されたラムダ式を使用しています。Java 7 以前であれば、リスト 2-3 のようになります。

リスト 2-2 実装詳細を隠蔽（Java 7 以前）

```
public final class X {
    ...
    public X() {
        ...       // 他の初期化処理
        Thread t = new Thread(new Runnable() {
            public void run() {
            ...  // スレッドでの処理
            }
        });
        t.start();
    }
}
```

練習問題 2.2：過去に書いたコードを見直して、Runnable インタフェースをトップレベルの public なクラスが実装していたり、Thread クラスをトップレベルの public クラスが継承していないかを調査しなさい。

2.3 パッケージ単位の公開 API の限界

Java 言語では、パッケージの下にさらにパッケージ、つまり、サブパッケージを作成することができます。しかし、あるパッケージとそのサブパッケージ間には特別な関係があるわけではありません。**どちらも独立したパッケージとしてアクセス制限が適用されます。**

大規模なソフトウェア開発では、あるパッケージを定義して、その中にすべての実装用のクラスやインタフェースを入れるのでなく、実装の一部を別のサブパッケージに定義したい場合があります。たとえば、`foo` パッケージの一部が `foo.bar` パッケージに定義されたとします。

これらの 2 つのパッケージは独立したパッケージとして扱われるために、`foo.bar` パッケージ内の公開 API は、設計上は `foo` パッケージからのみ使用されることを設計上は想定していても、それを強制する手段はありません。つまり、`foo` パッケージから使用したい `foo.bar` パッケージの機能はすべて `public` と宣言される必要があります。その結果、`foo.bar` パッケージ内の公開 API は、すべての他のパッケージにも公開されてしまうということになってしまいます。

この問題を解決するための方法としては、残念なら「運用」でパッケージの利用ルールを決めるというものがあります。たとえば、`foo` パッケージの実装詳細を構成するサブパッケージは、「`impl`」というサブパッケージ名を付けて `foo.impl` パッケージとするというものです。この場合何が「運用」部分かというと、`foo.impl` パッケージは「実装の詳細であり、公開されたものではなく、その内容は実装側の都合でいつでも自由に変更されるため、クライアントは使用してはならない」というルールを決めるものです。

これはあくまでも運用ルールなので、破ることは簡単にできてしまいます。そのため、OSGi（*Open Services Gateway initiative*）[*3] では、バンドル（*bundle*）単位でどのパッケージを他のバンドルからアクセス可能にするかを制御する機構を、独自のクラスローダを実装することで提供しています。また、Java 9 では、この問題を解決するための Java Platform Module System[*4] の検討が行われています。

2.4 まとめ

Java 8 までは、公開 API の単位はパッケージです。そのことを念頭に置いて API を設計する必要があります。他のパッケージから呼び出され、使用されることを想定してないクラス、インタフェース、メソッドなどは、「他のパッケージから呼び出してはならない」とドキュメンテーションに書くのではなく、最初から呼び出せないように設計しなければなりません。

[*3] `http://www.osgi.org`
[*4] `https://www.jcp.org/en/jsr/detail?id=376`

第 3 章

防御的プログラミング

3.1 防御的プログラミングとは

防御的プログラミング（*defensive programming*）とは、プログラミング上の誤りを早期に発見する手法の 1 つであり、主に、次の 2 つの検査を行うプログラミングです。

1. 公開 API である関数、コンストラクタ、メソッドなどの呼び出しで渡されるパラメータの値が、正しい範囲の値でないことを検査するプログラミング。
2. 設計上起きてはならない状態が発生した場合に、そのことを検査するプログラミング。

それぞれの検査で問題が発見された場合には、すみやかにシステムを停止させてデバッグする必要があります。検査で問題が発見された場合、どのソースコードを修正するかは、次の 2 つの場合に分かれます。

1. **呼び出し側**のソースコードを修正する必要がある場合（呼び出し側のバグ）。
2. **呼び出された側**のソースコードを修正する必要がある場合（呼び出された側のバグ）。

C/C++では、どちらの場合であっても、一般に `assert` を使用します。不正な値であったらデバッグ版では `assert` で停止するようにします。

Java 言語の場合、前者では、次のような例外をスローして、パラメータの値不正を通知します。後者では、`AssertionError` をスローします。

- `IllegalArgumentException`：`null` 値以外の引数不正
- `NullPointerExcetpion`：引数が `null` 値
- `IllegalStateException`：不正な状態での呼び出し

関数やメソッドなどのパラメータの不正な値を検査する防御的プログラミングが全く行われていない場合、不正な値が渡された場合には、次のようなことが発生します。

1. 不正な値での呼び出しが行われる。この時、内部的には不正な状態が発生する可能性がある。
2. そして、処理がそのまま続行される。
3. 処理がそのまま続行された結果、処理がかなり先に進んだ場所で不具合が発生する。

あるいは、

1. switch 文の case ラベルを追加しなければならない機能追加が行われた時に、一部の switch 文だけで追加を行い、他の switch への追加を忘れる。追加を忘れても、default ラベルで何もしないようにコードがもともと書かれている。
2. default レベルで処理がそのまま続行された結果、処理がかなり先に進んだ場所で不具合が発生する。

1 つ目の例では、不具合の最初の原因は、不正な値での呼び出しですが、不正であることを検査せずに処理を続行した場合には、全く関係のない箇所で不具合が発生したりします。この場合、不正な値での呼び出しと不具合の発生箇所が離れている場合には、最初の原因である不正な値での呼び出しまでさかのぼって調査するには、膨大な時間を要する場合があります。場合によっては、調査に数日要したりすることもあります。

ここで注意しなければならないのは、不正な値を検査してはいるが、assert で停止しなかったり、例外をスローしないで、単純にリターンしている場合には、防御的プログラミングを行っていることにはならないことです。この場合も、原因の調査に時間を要することになります。

一方、不正な値を検査する防御的プログラミングを行っている場合には、不正な値を検出した時点で、assert や例外によりシステムを即停止させますので、原因の調査を短時間に行うことができます。

2 つ目の例では、本来 default ラベルには設計上到達しないと思って何も書いていないことが原因です。つまり、設計上到達しないのであれば、防御的に assert で停止する必要があります。

3.2 防御的プログラミングをしない後ろ向きの理由

防御的プログラミングに関しては、Steve McConnell 氏の『Code Complete 第 2 版』の第 8 章「防御的プログラミング」や、Pete Goodliffe 氏の『Code Craft』の 1.5 節「防御的プログラミングとは何か」と 1.7 節「防御的プログラミングの技法」に書かれています。

今まで現場のソフトウェアエンジニアから聞いた、防御的プログラミングをしない「後ろ向きの理由」としては、次のようなものがあります。

1.「防御的プログラミング」って何ですか。習っていません。
   - 自分では勉強しないため、さまざまな本に書かれていることも、全部教えてもらえると思っている。
2. assert したり例外をスローしたりすると、呼び出し側が悪いのにもかかわらず、自分が不具合調査にいつも呼ばれるので、不正な呼び出しがあったり不正な状態になっても、何も検査しない。
   - プロジェクト全体の開発効率よりも、自分の工数を優先するエンジニア。
   - 自分が悪いかもしれないと全く考えないエンジニア。つまり、assert したり、例外をスローしたりしているコードの担当者に不具合の調査をたらい回しにするエンジニア。
3. assert したり、例外をスローしたりすると、システムが起動できないため、システムテストができないから外せと言われる。
   - システムテストを行える品質ではないにもかかわらず、計画されたシステムテスト開始日

になっても起動できないので、マネージャから外せと言われる。つまり、開発側の品質の作り込みよりも、システムテスト開始日を優先してしまうマネジメント。

防御的プログラミングを行う工数は、ほんの数分にもかかわらず、防御的プログラミングを行わなかったために、不具合調査に膨大な時間を要するようになり、結果としてプロジェクトを遅らせてしまいます。

私自身が防御的プログラミングを行うようになったのは、『Code Complete』の初版を読んでからだと思います。おそらく、1998 年か 1999 年の頃に読んだのだと思います。そのため、それ以前のプロジェクトであった *Fuji Xerox DocuStation IM 200* の開発では、防御的プログラミングを意識して行った記憶はありません。

本格的に防御的プログラミングを行うようになったのは、2000 年から Fuji Xerox 社で始まった、デジタル複合機のソフトウェアの再開発の時です。このプロジェクトでは、私自身が C++用のメモリ管理ライブラリ/スレッドライブラリ/コレクションライブラリを設計したこともあり、防御的プログラミングを、ライブラリの仕様や実装へ積極的に適用していました。また、いくつかの開発グループの設計レビューおよびコードレビューも行ったため、防御的プログラミングを推し進めていました。

また、2003 年から始まったあるプロジェクトは、ピーク時には 100 人程度のソフトウェアエンジニアが従事していましたが、防御的プログラミングを積極的に行うだけでなく、完全なテスト駆動開発も行っていました。

練習問題 3.1：今まで従事したソフトウェア開発プロジェクトでは、防御的プログラミングを行ってきましたか。行っていないとしたら、その理由は何ですか。

練習問題 3.2：第 1 章の C 言語でのスタックの実装例では、言及していませんが、防御的プログラミングが行われています。みなさんが作成した練習問題 1.4 の C++言語版の解答では、防御的プログラミングを行いましたか。

3.3 パラメータ値不正

公開 API における不正なパラメータ値に対する防御的プログラミングの第 1 歩は、**防御的にプログラミングされている事実が、公開 API の仕様に明確に記述されている**ことです。言い換えると、公開 API には不正なパラメータ値に対する動作が明確に記述されている必要があります。

3.3.1 C/C++

C/C++の場合には、基本的に `assert` を使用します。リスト 3-1 は、ツリーマップの公開ヘッダーファイルの例です。

リスト 3-1 `treeMap.h`

```
struct treeMap; /* 前方宣言 */
```

```
/**
 * ツリーマップを作成して返します。
 */
struct treeMap *treeMap_create(void);

/**
 * ツリーマップを解放します。
 *
 * map が NULL であれば、デバッグ版では assert で停止します。
 * リリース版では何もしないでリターンします。
 */
void treeMap_delete(struct treeMap *map);

/**
 * ツリーマップへキーと値の組を入れます。指定されたキーがすでに存在すれば、
 * 指定された値で置き換えます。値として NULL（0）を入れることはできません。
 *
 * map が NULL であれば、デバッグ版では assert で停止します。
 * リリース版では何もしないでリターンします。
 *
 * key が NULL であれば、デバッグ版では assert で停止します。
 * リリース版では何もしないでリターンします。
 *
 * value が NULL であれば、デバッグ版では assert で停止します。
 * リリース版では何もしないでリターンします。
 */
void treeMap_put(struct treeMap *map, char *key, void *value);

/**
 * ツリーマップから指定されたキーを取り除き、それに関連付けられていた値を
 * 返します。指定されたキーが存在しない場合には、NULL が返されます。
 *
 * map が NULL であれば、デバッグ版では assert で停止します。
 * リリース版では何もしないでリターンします。
 *
 * key が NULL であれば、デバッグ版では assert で停止します。
 * リリース版では何もしないでリターンします。
 */
void *treeMap_remove(struct treeMap *map, char *key);

/**
 * ツリーマップから指定されたキーに関連付けされている値を返します。
 * 指定されたキーが存在しない場合には、NULL が返されます。
 *
 * map が NULL であれば、デバッグ版では assert で停止します。
```

```
 * リリース版では何もしないでリターンします。
 *
 * key が NULL であれば、デバッグ版では assert で停止します。
 * リリース版では何もしないでリターンします。
 */
void *treeMap_get(struct treeMap *map, char *key);
```

リスト 3-1 では、不正なパラメータ値に対する動作が仕様として記述されています。ここで注意しなければならないのは、デバッグ版とリリース版の 2 種類のオブジェクトが生成されることを前提としていることです。

通常、assert.h をインクルードすることにより使用される assert は、デバッグ用コンパイルとリリース用コンパイルで異なる動作をします。そのため、通常のビルドでは、リリース版とデバッグ版の両方を作成するようにします。そして、開発部門では、通常はデバッグ版を使用して開発を行います。

実装用のヘッダーファイルは、リスト 3-2 のようになります。

リスト 3-2 treeMapImp.h

```
struct treeMap {
    struct treeMap *left;
    struct treeMap *right;
    char           *key;
    void           *value;
};
```

実装は、リスト 3-3 のようになります。

リスト 3-3 treeMapImp.c

```
#include <assert.h>
#include <stdlib.h>

#include "treeMap.h"
#include "treeMapImpl.h"

struct treeMap *treeMap_create(void) {
    struct treeMap *map = (struct treeMap *) malloc(sizeof(struct treeMap));
    if (map == NULL) {
        assert(0 && "map cannot be created");
        return NULL;
    }
```

```
    map->left  = NULL;
    map->right = NULL;
    map->key   = NULL;
    map->value = NULL;
    return map;
}

void treeMap_delete(struct treeMap *map) {
    if (map == NULL) {
        assert(0 && "illegal argument");
        return;
    }

   ...  /* すべてのデータを消去*/
    free(map);
}

void treeMap_put(struct treeMap *map, char *key, void *value) {
    if (map == NULL || key == NULL || value == NULL) {
        assert(0 && "illegal argument");
        return;
    }

    ... /* キーと値を追加 */
}

void *treeMap_remove(struct treeMap *map, char *key) {
    if (map == NULL || key == NULL) {
        assert(0 && "illegal argument");
        return NULL;
    }

    ... /* キーを探して取り除き、その保存してあった値を返す */
}

void *treeMap_get(struct treeMap *map, char *key) {
    if (map == NULL || key == NULL) {
        assert(0 && "illegal argument");
        return NULL;
    }

    ... /* キーを探して、その対応する値を返す */
}
```

ここで、リスト 3-4 のような実装をしないようにしなければなりません。

リスト 3-4 誤った実装

```
void treeMap_put(struct treeMap *map, char *key, void *value) {
    assert((map == NULL || key == NULL) && "illegal argument");

    ... /* キーと値を追加 */
}
```

この実装では、リリース版で不正な値が渡された時に、assert が無視されて、そのまま処理が続いてしまいます。

リスト 3-3 で、重要な点として、コードを見ただけで、何が不正な値なのかが明確に分かります。なぜなら、assert が使われているからです。一方、リスト 3-5 のようなコードであればどうでしょうか。

リスト 3-5 パラメータの不正値が不明

```
void treeMap_put(struct treeMap *map, char *key, void *value) {
    if (map == NULL || key == NULL) {
        return;
    }

    ... /* キーと値を追加 */
}
```

リスト 3-5 の場合、map や key に NULL を渡すことができるという仕様なのか、本当は不正だけど、念のために検査しているのかが、明確には読み取れません。

練習問題 3.3：今までの練習問題のプログラミングで Makefile を作成していなければ、作成しなさい。そして、デバッグ版とリリース版の両方が同時に作成されるように Makefile を工夫しなさい。リリース版のコンパイルでは、コンパイルオプションとして-DNDEBUG を指定します。

練習問題 3.4：みなさんが従事した C/C++の開発プロジェクトで、リリース版とデバッグ版の両方を同時にビルドするようになっていたプロジェクトはありましたか。

3.3.2 Java

Java の場合には、基本的に例外をスローします。リスト 3-1 のツリーマップを Java で書き直したのがリスト 3-6 です。

リスト 3-6 TreeMap.java

```
import java.util.Objects;

public final class TreeMap<K, V> {

    public TreeMap() {
        // ...
    }

    /**
     * ツリーマップへキーと値の組を入れます。指定されたキーがすでに存在すれば、
     * 指定された値で置き換えます。
     *
     * @param key キー
     * @param value 値
     * @throws NullPointerException key が null の場合
     */
    public void put(K key, V value) {
        if (key == null)
            throw new NullPointerException("key is null");

        // ...
    }

    /**
     * ツリーマップから指定されたキーを取り除き、それに関連付けられていた値を
     * 返します。指定されたキーが存在しない場合には、null が返されます。
     *
     * @param key キー
     * @return キーに関連づけされていた値、もしくは、キーが存在しなければ null
     * @throws NullPointerException key が null の場合
     */
    public V remove(K key) {
        Objects.requireNonNull(key, "key is null"); // Java 7 以降

        // ...
    }

    // ...
}
```

リスト 3-6 で重要な点は、以下の通りです。

- key として null は許されないことが、@throws タグを使用して Javadoc に書かれている。

- NullPointerException 例外にメッセージがきちんと渡されている(NullPointerException 例外のメッセージが空文字列の場合には、通常はパラメータ値不正なのか呼び出された側のバグなのか区別できません)。

null に対しては、慣例として IllegalArgumentException ではなく NullPointerException をスローします。それ以外の、パラメータの不正値に対しては、IllegalArgumentException を使用します。また、Java 7 以降であれば、ユーティリティメソッドとして提供されている Objects.requireNonNull() メソッドを使用してください。

3.4 呼び出し順序不正

関数やメソッドに呼び出し順序の制約がある場合があります。たとえば、close() 関数（メソッド）は、open() 関数（メソッド）を呼び出して成功した場合にしか呼び出してはならないというものです。**API 仕様書には、そのような呼び出し順序制約をきちんと記述しなければなりません。**しかし、さらに重要なのは、**不正順序呼び出しに対して防御的プログラミングを行い、不正順序呼び出しの場合の振る舞いも記述する**ことです。

呼び出し順序不正に関しては、次のような対処となります。

- C/C++の場合には、assert を用いますが、パラメータ不正と同様に、デバッグ版とリリース版の振る舞いも記述します。
- Java の場合には、IllegalStateException 例外をスローします。

呼び出し順序不正を検出するには、当然、現在の状態を管理するための変数やフィールドが追加で必要となります。それでも、防御的にプログラミングすることは必要です。

練習問題 3.5：今までのソフトウェア開発において、呼び出し順序制約を仕様にきちんと書いてきましたか。もし、書いたことがないとしたら、書かなかった理由を述べてください。その理由は、実際に間違った呼び出しが行われた時に発生するバグを効率的に発見することを妨げていませんでしたか。

3.5 設計ロジック誤り

防御的プログラミングでの最後の例は、設計ロジックの誤りです。つまり、設計上、絶対に到達することがない箇所に到達したら、それは設計ロジックの誤りですので、検出するためのプログラミングを行う必要があります。そして、呼び出された側で誤りを修正する必要があります。

3.5.1 switch 文における default ラベル処理

防御的プログラミングの観点から、switch 文では原則次のようにコードを書く必要があります。

- default ラベルを省略せずに書く。

- default ラベルでの処理としては、C/C++では必ず assert を書き、Java では AssertionError をスローする。

この 2 つを守るには、通常は、すべての値を case ラベルで列挙する必要があります。次のコードは、『Effective Java 第 2 版』からの抜粋です。

リスト 3-7 Operation.java

```
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    // 定数で表される算術操作を行う
    double apply(double x, double y) {
        switch(this) {
            case PLUS:   return x + y;
            case MINUS:  return x - y;
            case TIMES:  return x * y;
            case DIVIDE: return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

この例では、default ラベルは使用していませんが、switch 文のどの case ラベルにも該当しなければ、AssertionError がスローされています。ここで重要なのは、設計段階で到達しない箇所に到達したらスローされるように書かれていることです。言い換えると、Operation が定義している 4 つの enum 値はすべて case ラベルで列挙されていますので、絶対に到達しないわけです。

では、実際にはいつ到達するかというと、「将来、誰かが定数値を増やしたけど、そのための case ラベルを追加するのを忘れた時」です。つまり、将来の修正に対して防御的にプログラミングしていることになります。

switch 文だけでなく、if-else の連なりの場合も、最後の else は到達することがないという前提でコードを書く必要があります。

3.5.2 設計上到達しない

設計上到達しない例として、次のコードを見てみてください。

リスト 3-8 設計上到達しない（悪い例）

```
private Object findValueFromTable(Object key) {
    for (int i = 0; i < table.length; i++) {
```

```
            if (table[i].key().equals(key))
                return table[i].value();
        }
        return null;
    }
```

この findValueFromTable() メソッドの呼び出しは実装の詳細であり、必ずインスタンスフィールドである table 内に該当データがあるという設計だとします。つまり、for 文が終了して、return null; の文が実行されことは設計上ないとします。この場合、findValueFromTable() メソッドは戻り値を返すと宣言されているために何らかの値を返すコードを書く必要があり、return null; と書いているわけです。

しかし、このようなコードを見た時に、null を返すのが正常な処理なのか、それとも、コンパイルエラーを回避するために return 文を書いているのか区別できません。したがって、次のように書かなければなりません。

リスト 3-9 設計上到達しない（正しい例）

```
    private Object findValueFromTable(Object key) {
        for (int i = 0; i < table.length; i++) {
            if (table[i].key().equals(key))
                return table[i].value();
        }
        throw new AssertionError("impossible case");
    }
```

こうすれば、for 文のループ中で return 文が実行されることなく、for 文が終了することそのものが、本来発生してはいけないことが明白になります。

このような場合、C/C++では、assert 文の後に return 文を書いて、何らかの値を返す必要があります。

3.6 assert 関数マクロについて

ここでは、C/C++で用いる assert 関数マクロに関する注意事項を述べます。

- システム標準の関数マクロ assert は、リリース版では取り除かれてしまう。
- 品質保証部門（QA）や市場での障害を調査するためには、完全に取り除くのではなく、別の関数マクロとして展開されて、何らかの記録（ログ）が残される必要がある。
- システム標準の関数マクロ assert は、デバッグ版ではコアダンプしたりするようになっている。システム標準を使用せずに、独自の関数マクロ assert を用意した方がよい。その場合、

デバッグ版ではスタックトレースなどの情報を出力した後にスレッドをサスペンドし、リリース版ではログを記録するようにする。

- リリース版での記録は、システムがブートしてからの最初の数個（10 個程度）を記録しておけばよい。

システムがブートしてから、一度 assert の状態が発生すると、その呼び出し元でも assert が発生するという「assert の連鎖」が発生する可能性が高いです。その結果、リリース版では、すべてを記録しても役に立たないし、最後の 10 個だけを記録しても役に立たない可能性が非常に高いです。したがって、システムがブートしてから発生した最初の数個を記録しておくだけで十分です。

> ここで述べていることは、2000 年の頃にある商品開発の際に考えて、実践したものです。

3.7　キャッチされない例外をキャッチする

Java での防御的プログラミングでは、基本的に例外をスローします。通常、そのような例外は誰もキャッチしませんので、該当スレッドは終了することになります。しかし、システムとしては、後の調査のために何らかのログを残す必要があります。

詳細は説明しませんが、Java 5.0 以降であれば、システム全体のスレッドに対して、そのような例外用のハンドラを Thread.setDefaultUncaughtExceptionHandler() で登録することができます。それより前の Java のバージョンでは、ThreadGroup を使用する必要があります [*1]。

3.8　コードカバレッジと防御的プログラミング

テストコードによるコードカバレッジは、そのソフトウェアの品質を担保しません。コードカバレッジが 100%だから、品質が高いとは限りません。バグがあるコードでも、テストでコードカバレッジを 100%にすることはできます。

一方、この章で述べた防御的プログラミングを行うと、通常のテストでは、コードカバレッジを 100%にすることはできません。引数の不正値に関してはテストできますが、設計上到達しないコードはテストできません。結果として、100%にはならないわけです。

[*1] 柴田芳樹、「Uncaught Exception をキャッチする」、*Java PRESS, Vol.12*、2000 年 5 月、技術評論社

付録 A

防御的プログラミングに関して

（Peter Goodliffe 著の『Code Craft – エクセレントなコードを書くための実践的技法 –』からの抜粋（p.26）です。ボールドは私自身が付けたものです。）

単なる正しいコードではなく、正しく、しかも優れたコードを作り上げることが重要です。想定されている前提条件はすべて文書化する必要があります。このことは、コードの保守を容易にして、コード内に潜在するバグを減らすことにつながります。**防御的プログラミング**は、最悪の事態を予期して、それに備えるための手段です。この技法を使用すると、単純な欠陥がやがて検出しにくいバグになることを回避できます。

コードとして書き表した制約を防御的なコードと共に活用すると、ソフトウェアの堅牢性が格段に向上します。その他の数多くのコーディング技法と同様に、防御的プログラミングは、初期の段階で少しだけ余計に時間をかけて有効な作業を行うことによって、後の段階での時間と労力とコストの大幅な削減を実現するという考え方に基づいています。その効果は、掛け値なしに、プロジェクト全体を破滅から救えるほどに大きいものだと**断言できます**。

良いプログラマーは

- コードの堅牢性を重視する
- すべての前提条件が防御的コードに明示的に反映されるように常に心掛ける
- 無効なデータが入力された場合の動作を明確に定義することを望む
- 自分が書いているコードについて注意深く考えながらコードを書く
- 他人（または自分自身）の愚かさに対する防御を備えたコードを書く

悪いプログラマーは

- コードで起こり得る問題については考えようとしない
- ほかのコードと結合されるコードを、エラーが発生する可能性のある状態のまま引渡し、誰かが問題を解決してくれることを期待する
- 自分が書いたコードの適切な使用法に関する重要な情報を自分の頭の中にしまい込んで、いつ忘れてもおかしくない状態のまま放置する
- 自分が書いているコードについて深く考えることをせず、結果的に、予期しない動作がたびたび発生する不安定なソフトウェアを作成してしまう

■ 著者紹介

柴田　芳樹 (しばた　よしき)：1959 年生まれ。九州工業大学情報工学科で情報工学を学び、1984 年同大学大学院で情報工学修士課程を修了。Unix（Solaris/Linux）、C、Mesa、C++、Java、Go などを用いた様々なソフトウェア開発に従事。米国ゼロックス社のパロアルト研究所を含め、カリフォルニア州（El Segundo 市、Palo Alto 市）およびニューヨーク州（Webster 市）にて米国ゼロックス社でのソフトウェア開発を経験。私的な時間に専門書の翻訳を行っている。現在は、ソフトウェアエンジニアとして教育、コンサルテーション、ソフトウェア開発に従事。

訳書

『プログラミング言語 Go』『Effective Java 第 2 版』（以上、丸善出版）、『Java SE 8 実践プログラミング』『API デザインの極意』（以上、インプレス）、『プログラミング言語 Java 第 4 版』『プログラミング原論』（以上、東京電機大学出版局）、『アプレンティスシップ・パターン』（オライリー・ジャパン）、『プログラミング言語 Go フレーズブック』『Objective-C 明解プログラミング』『Android SDK 開発クックブック』『Java Puzzlers 罠、落とし穴、コーナーケース』『Google Web Toolkit ソリューション』『Java リアルタイム仕様』（以上、ピアソン桐原）

著書

『Java 2 Standard Edition 5.0 Tiger』（ピアソン・エデュケーション）
『ソフトウェア開発の名著を読む【第二版】』（技術評論社）
『プログラマー”まだまだ”現役続行』（技術評論社）